Università degli Studi di Parma
Dipartimento di Ingegneria e Architettura
Sistemi operativi e in tempo reale – a.a. 2022/23

Monitor: un *paradigma* di programmazione concorrente in ambiente globale

prof. Stefano Caselli
stefano.caselli@unipr.it

Oltre i semafori

- I semafori sono considerati uno strumento potente ma di basso livello: l'*assembly della concorrenza*
- Problema: scarsa scalabilità in problemi poco più grandi degli esempi visti
  - Esistono alcuni *pattern di uso dei semafori* per problemi canonici, ma in generale i problemi reali restano complicati da risolvere mediante semafori o variabili di lock
- Una delle fonti di complessità è il fatto che il semaforo venga utilizzato per *due scopi*: realizzare la *mutua esclusione* e stabilire le *condizioni di progresso dei thread*
  - Esempio problema Produttore-Consumatore: scambiare l'ordine delle wait() può provocare deadlock, e questo rischio non è immediatamente percepibile

Oltre i semafori: altre dimensioni di discussione

- *Programming in the large* vs. *Programming in the small*
  - sviluppare grandi sistemi concorrenti con i semafori è improponibile
  - nel tempo sono state sviluppate metodologie per progettare grandi sistemi software, tra cui un passaggio importante (attorno al 1980) è stata l'idea della programmazione ad oggetti
  - come si coniuga la concorrenza con la programmazione a oggetti?
- Esiste un problema di astrazione e hiding nella programmazione concorrente?
  - Sì, usando i semafori il dettaglio della sincronizzazione è esposto in tutti i thread, con possibili inconsistenze
- Sono stati proposti altri meccanismi che forniscono maggiore astrazione, anche a costo di prestazioni inferiori

Oltre i semafori: il Monitor

- Obiettivo: un *meccanismo di sincronizzazione più leggibile dal programmatore*, che consenta di controllare chi risvegliare tra i thread in attesa, con un compromesso equilibrato o comunque regolabile tra astrazione ed efficienza
- Il **Monitor** è un *paradigma di programmazione concorrente*
- Può essere offerto nativamente da un linguaggio di programmazione (ad es. Java)
- Oppure, in molti casi, essere realizzato con meccanismi di più basso livello forniti dal SO, come semafori, lock e altro

Oltre i semafori: il Monitor

- Il Monitor prevede di definire una sincronizzazione mediante un meccanismo separato che realizza la **mutua esclusione** (può essere un semaforo mutex o un lock) e un meccanismo ad hoc per ritardare i thread che non possono eseguire: la **variabile condizione**
- In generale è possibile prevedere *variabili condizione distinte* per separare le condizioni logiche di attesa
- In qualche caso il linguaggio o la libreria prevede *un'unica variabile condizione* per tutti i thread che devono attendere
- Un thread verifica la propria condizione logica di progresso detenendo il mutex o il lock, e lo rilascia automaticamente se deve sospendersi

Monitor definito all'interno di un linguaggio

```
type <nome del tipo> = monitor {
  <dichiarazione di variabili locali>
procedure entry <nome1>( ... );
  begin  ...   end;
procedure entry <nome2>( ... );
  begin  ...   end;
procedure <nome3> ( ... );
  begin  ...   end;
procedure <nome4> ( ... );
  begin  ...   end;

begin <statement iniziale> end;
}
```

Keyword *monitor*: identifica definizione di un tipo di dato astratto (ADT) per sincronizzazione

E' un *oggetto astratto condiviso* definito da struttura dati, procedure entry e non entry, inizializzazione dello stato (come in ADT)

A questi elementi si aggiunge la specifica della sincronizzazione

Le istanze di un monitor sono accessibili contemporaneamente da più thread!!
→ problemi di consistenza e ordinamento degli eventi di thread diversi

Monitor non supportato dal linguaggio

- Se il linguaggio non offre nativamente il costrutto Monitor o simile (ad es. in Java: ogni oggetto può essere utilizzato come Monitor proteggendolo con la parola chiave *synchronized*) è possibile adottare il Monitor come pattern, realizzandolo con i meccanismi di sistema (es. libreria POSIX per multithreading)
- C e C++ non offrono il Monitor come costrutto linguistico: nelle esercitazioni realizzerete sincronizzazioni basate su Monitor mediante funzioni delle API POSIX e altre funzionalità del C++

Monitor

- Le istanze di un monitor sono oggetti condivisi e quindi accessibili contemporaneamente da più thread
- → per mantenere la consistenza della struttura dati del monitor ed evitare interferenze, le *procedure entry* devono essere eseguite in *mutua esclusione*
- Nei linguaggi che offrono nativamente il costrutto Monitor tale *regola di sincronizzazione* è garantita dal *compilatore*
- Se la sincronizzazione è realizzata mediante API di libreria (es. POSIX), è il programmatore che inserisce un *mutex* o un *lock* seguendo il *monitor come pattern*

Monitor

- Scopo del monitor è *controllare l'assegnazione* di una risorsa a thread concorrenti in base a determinate politiche di gestione
- L'assegnazione avviene secondo *due livelli*:
  1. *Garanzia che un solo thread alla volta abbia accesso al monitor*

     → Le procedure del monitor sono eseguite da un thread alla volta
  2. *Controllo dell'ordine con cui i thread hanno accesso alla risorsa*

     In funzione dello stato della risorsa, la procedura può sospendere il thread in una coda locale al monitor; la sospensione comporta la *liberazione del monitor*

Monitor

- Controllo dell'assegnazione della risorsa protetta dal Monitor:

- Per quanto riguarda il 1° livello, la soluzione è la presenza di un *unico* semaforo mutex (v.i. = 1) per proteggere *ogni procedura entry* del monitor. L'accesso al monitor è *serializzato*
- Per quanto riguarda il 2° livello, si introducono *nuove variabili* denominate *variabili di tipo condizione*

Variabili di tipo condizione

- Ogni variabile di tipo condizione rappresenta *una coda* nella quale possono sospendersi i thread
- Una variabile condizione viene realizzata mediante un campo valore ed un campo puntatore (alla coda)
- Il programmatore può prevedere *tante variabili di tipo condizione* quante sono le *condizioni logiche distinte* per cui un thread può essere ritardato
- Le condizioni logiche che interessa distinguere possono essere *più o meno specifiche*. Nel caso limite, ci può essere una condizione per ogni thread (con un'architettura analoga a quella dei *semafori privati*)

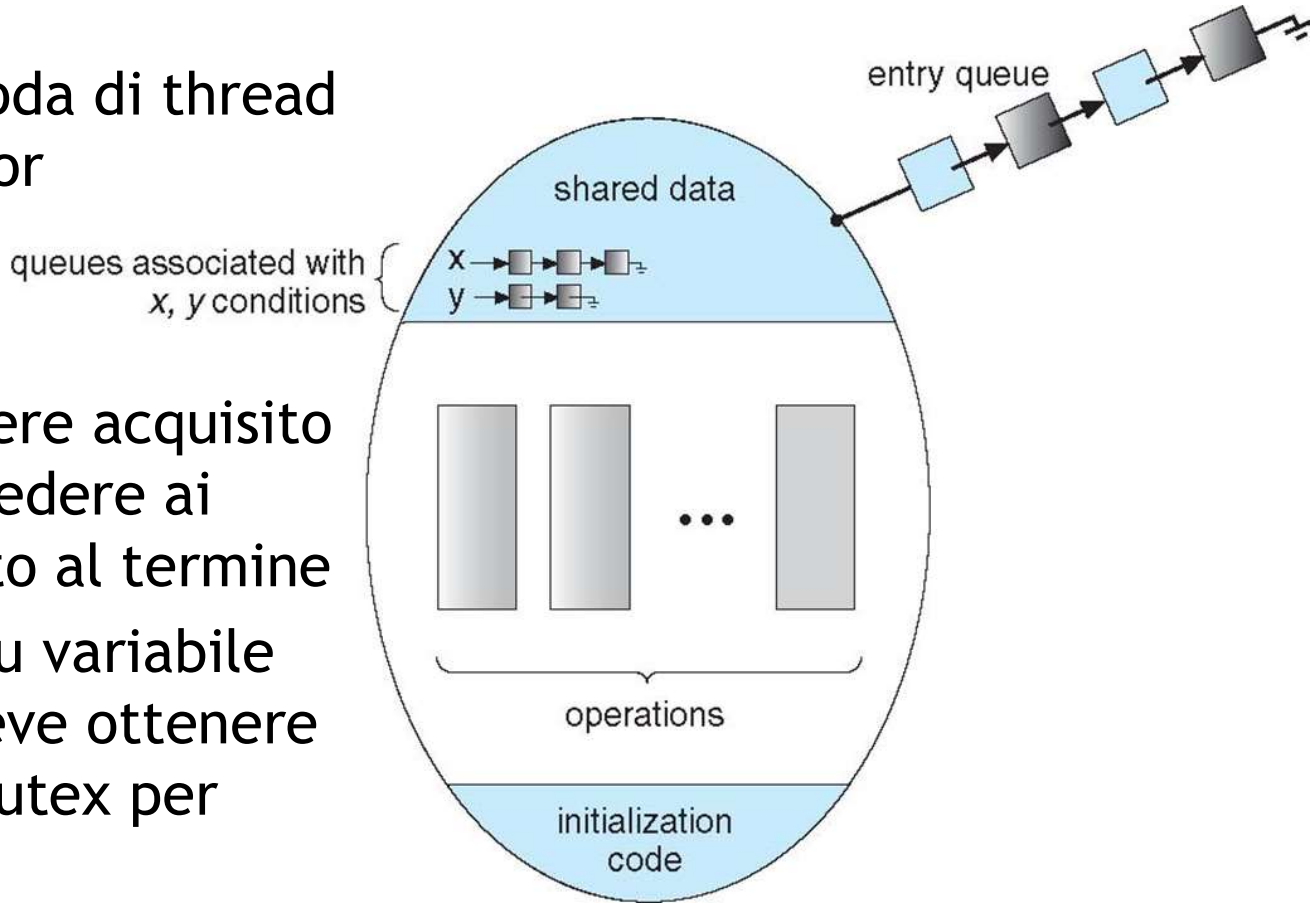
Variabili di tipo condizione

- Le procedure del monitor agiscono sulle variabili condizione tramite le operazioni **cond.wait** e **cond.signal**
- denominazione alternativa: *cond.delay* e *cond.continue*

- L'esecuzione della operazione **cond.wait** *sospende sempre il thread* e lo inserisce nella coda associata alla variabile *cond*. La sospensione libera il monitor, cioè il mutex è rilasciato
- L'esecuzione della operazione **cond.signal** *rende attivo un thread* in attesa nella coda individuata dalla variabile *cond*. Nel caso di coda vuota *non ci sono effetti collaterali*
- La semantica della cond.signal è da approfondire!!

Monitor e variabili condizione

- Lock o semaforo mutex: protezione ai dati condivisi
- Variabile condizione: coda di thread in attesa entro il monitor



- Il lock deve sempre essere acquisito dal thread prima di accedere ai dati condivisi e rilasciato al termine
- Anche dopo un'attesa su variabile condizione, il thread deve ottenere nuovamente il lock o mutex per accedere

Esempio: Produttori - Consumatori

- Progettiamo un monitor per risolvere un problema di Produttori-Consumatori --> inizialmente definiamo un ADT

  type mailbox = *monitor* ...

- Dichiarazioni delle variabili:

```
var  a, b:   messaggio;    // var del tipo dei valori scambiati
var  B, D:   mailbox;      // creazione istanze del Monitor
```

- Uso del monitor B per sincronizzazione dei thread P e C:

```
P:   B.send(a);            // le operazioni definite dal tipo
C:   B.receive(b);         // sono denominate send e receive
```

Esempio: Produttori - Consumatori

```
type  mailbox = monitor {
        var     buffer: array [0 .. N-1] of  messaggio;   /* rappr.  oggetto  */
                testa, coda: 0 .. N-1;
                cont: 0 .. N;
                non_pieno, non_vuoto: condition;
 procedure entry  send (x: messaggio);
        begin
                if  cont = N  then  non_pieno.wait;
                buffer [coda] := x;
                coda := (coda + 1) mod  N;
                cont := cont + 1;
                non_vuoto.signal
        end
```

le due condizioni per attesa di P e C

thread attende se cont=N

deve essere risvegliato con condizione cont<N valida

risveglio di eventuale consumatore in attesa

Esempio: Produttori - Consumatori

```
procedure entry  receive (var  x: messaggio);
        begin
                if  cont = 0 then non_vuoto.wait;
                x := buffer [testa];
                testa := (testa + 1) mod  N
                cont := cont - 1;
                non_pieno.signal
        end
begin  cont := 0; testa := 0; coda := 0  end
}               /* fine definizione del tipo */
```

il thread attende se buffer vuoto

quando è risvegliato ci deve essere elemento

ha liberato risorsa, eventuale risveglio di produttore

Esempio: Produttori - Consumatori

- Soluzione del problema Produttori-Consumatori mediante un'istanza del monitor mailbox:

var a, b: messaggio;
B: mailbox;

Pi	Ci
...	...
B.send(a);	B.receive(b);
...	...

- La sincronizzazione dei thread non scompare (come con altri meccanismi) ma è incapsulata nella definizione del Monitor

Monitor per Produttori-Consumatori

- Nell'ambito dei thread Pi e Ci scompare il riferimento alle primitive di sincronizzazione di basso livello wait() e signal(). Le primitive sono confinate e mascherate dalle procedure send() e receive(), scritte *una volta per tutte* all'interno del monitor

- *(Maybe it's cheap) but it ain't free...*
- Rispetto alla soluzione con i semafori, qui non è più possibile inserire e prelevare *in parallelo* su posizioni diverse: il compilatore (o il pattern) introduce *un unico semaforo mutex* per tutte le procedure del monitor, impedendo accessi contemporanei sull'intera risorsa

Monitor Produttori-Consumatori

- Esercizio
- Cosa accade con:

```
var     a, b: messaggio;
        B, D: mailbox;
```

T1	T2
...	...
B.send(a);	D.receive(b);

Monitor con supporto linguistico

- Il compilatore realizza direttamente la serializzazione (mutex o lock non visibili dal programmatore)
- Realizza le variabili condizione con meccanismi di base forniti dal sistema operativo
- In genere il Monitor è realizzato in linguaggi che forniscono incapsulamento del dato condiviso mediante Tipo di Dato Astratto o Oggetto: --> definizione di tipo o classe e creazione di istanza
- Alcuni linguaggi storici: Concurrent Pascal, Modula, Mesa, Java

Monitor in assenza di supporto linguistico

```
lock buf_lock = <initially unlocked>
condition producer_CV = <initialization function>
condition consumer_CV = <initialization function>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) { cond_wait(&producer_CV, &buf_lock); }
  enqueue(item);
  cond_signal(&consumer_CV);
  release(&buf_lock);
}
Consumer() {
  acquire(buf_lock);
  while (buffer empty) { cond_wait(&consumer_CV, &buf_lock); }
  item = dequeue();
  cond_signal(&producer_CV);
  release(buf_lock);
  return item
}
```

Monitor in assenza di supporto linguistico

- Molti dei linguaggi in uso attualmente e dei nuovi linguaggi proposti utilizzano il *monitor come pattern*
  - Maggiore flessibilità nella realizzazione come libreria e adattamento più semplice alle diverse piattaforme hw
- La sincronizzazione dei thread è realizzata con mutex e variabili condizione o con semafori, utilizzati in stile monitor
- I meccanismi di base sono forniti da una libreria di multithreading (come POSIX)

Uso delle variabili condizione: alternative

- Variabili condizione come *code monothread* (una condizione per ogni thread, come semafori privati): il programmatore ha il *controllo completo* sulla scelta del thread da risvegliare
  - Occorre conoscere *a priori* quanti thread agiscono sulla risorsa per inserire nella definizione del tipo le relative variabili condizione
  - Ogni thread, quando esegue una procedure del monitor, specifica la propria identità con un indice
- Altre *funzioni primitive* su variabili di tipo condizione:

  cond.queue
  - Verifica la presenza nella coda *cond* di almeno un thread sospeso
  - Esempio: *if* cond.queue *then* ...

Esempio: Produttori - Consumatori

```
type  mailbox = monitor {
        var      buffer: array [0 .. N-1] of  messaggio;   /* rappr.  oggetto  */
                 testa, coda: 0 .. N-1;
                 cont: 0 .. N;
                 non_pieno, non_vuoto: condition;
 procedure entry  send (x: messaggio);
        begin
                 if  cont = N  then  non_pieno.wait;
                 buffer [coda] := x;
                 coda := (coda + 1) mod  N;
                 cont := cont + 1;
                 non_vuoto.signal
        end
```

thread attende se cont=N

deve essere risvegliato con condizione cont<N valida

risveglio di eventuale consumatore in attesa

Esempio: Produttori - Consumatori

```
procedure entry  receive (var  x: messaggio);
        begin
                if  cont = 0 then non_vuoto.wait;
                x := buffer [testa];
                testa := (testa + 1) mod  N
                cont := cont - 1;
                non_pieno.signal
        end
  begin  cont := 0; testa := 0; coda := 0  end
}               /* fine definizione del tipo */
```

il thread attende se buffer vuoto

quando è risvegliato ci deve essere elemento

ha liberato risorsa, eventuale risveglio di produttore

Esempio: Lettori-Scrittori

```
type  lettori_scrittori = monitor {
 var    num_lettori: integer; occupato: boolean;   /* N. lettori su risorsa */
        ok_lettura, ok_scrittura: condition;        /* e presenza scrittore */
 procedure entry  Inizio_lettura;
  begin
        if  (occupato or ok_scrittura.queue) then  ok_lettura.wait;
        num_lettori := num_lettori + 1;
        ok_lettura.signal;              /* risveglio a catena dei lettori già nel  */
  end                                   /* monitor dopo accesso di scrittore    */
procedure entry  Fine_lettura;
 begin
        num_lettori := num_lettori - 1;
        if  num_lettori = 0 then  ok_scrittura.signal;
 end
```

Esempio: Lettori-Scrittori

```
procedure entry  Inizio_scrittura;
 begin
        if ((num_lettori <> 0) or occupato) then  ok_scrittura.wait;
        occupato := true;
 end
 procedure entry  Fine_scrittura;
 begin
        occupato := false;
        if  ok_lettura.queue then  ok_lettura.signal;
                          else  ok_scrittura.signal;
 end
 begin  num_lettori := 0;    occupato := false;  end
}
```

Soluzione *weak-weak writer preference*!

Sostituendo con:
'if ok_scrittura.queue then ok_scrittura.signal; else ok_lettura.signal;'
si ha *strong writer preference*

Filosofi a cena: soluzione mediante *monitor*

- Definizione del tipo: *type* dining-philosopher = *monitor* { ... }
- Creazione dell'istanza: *var* dp: dining-philosopher;

Philosopher_i:

repeat

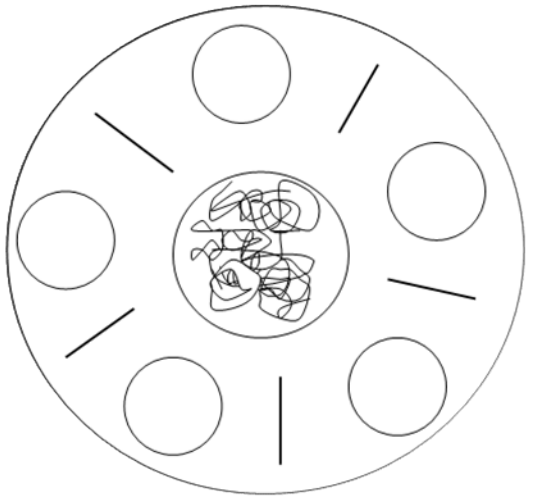
<think>;

dp.pickup(i);

<eat>;

dp.putdown(i);

forever



Filosofi a cena: soluzione mediante *monitor*

```
type  dining-philosopher = monitor {
  var  state:  array  [0 .. N - 1]  of   (thinking, hungry, eating);
       self:   array  [0 .. N - 1]  of   condition;

  procedure entry  pickup (i:  0 .. N-1);
  procedure entry  putdown (i:  0 .. N-1);

  begin  for  i := 0  to  N-1 do  state [i] := thinking;  end
} //end type
```

- La definizione del tipo di dato astratto include sempre anche l'inizializzazione dello stato

Filosofi a cena: soluzione mediante *monitor*

```
type  dining-philosopher = monitor
 var  state:  array  [0 .. N - 1]  of   (thinking, hungry, eating);
      self:   array  [0 .. N - 1]  of   condition;

 procedure entry pickup (i:  0 .. N-1)  {
                state [i] := hungry;
                if  (state [(i+N-1)mod N] <> eating
                     and state [(i+1)mod N] <> eating)
                then  state [i] := eating;
                else   self [i].wait;
 }
```

Filosofi a cena: soluzione mediante *monitor*

```
procedure entry putdown (i: 0 .. N-1)  {
        state [i] := thinking;
        if  state [(i+N-1)mod N] = hungry and state [(i+N-2)mod N] <> eating
        then  begin
                state [(i+N-1) mod N] := eating;
                self [(i+N-1) mod N] .signal;
              end
        if  state [(i+1) mod N] = hungry and  state [(i+2) mod N] <> eating
        then  begin
                state [(i+1) mod N] := eating;
                self [(i+1) mod N] .signal;
              end
}
```

Filosofi a cena: soluzione mediante *monitor*

- Discussione:
- Soluzione analoga a quella basata su semafori privati; prevede una variabile condizione per ogni filosofo
- Queste soluzioni di sincronizzazione sono adottabili solo a problemi con numero di thread noto e costante per la applicazione
- E' possibile realizzare una soluzione mediante Monitor al problema dei filosofi che preveda un'unica variabile condizione? Sotto quali ipotesi?

Tipi di monitor: semantiche alternative

- Signal and Wait: Hoare
- Signal and Exit: Brinch-Hansen
- Signal and Continue: Mesa

Tipi di monitor

- Ogni monitor rafforza e mantiene un *invariante* di correttezza dello stato
  - ad esempio: $0 \leq d\text{-}e \leq N$ nel problema produttori-consumatori con buffer limitato
- Ogni volta che il monitor è liberato o commutato l'invariante deve essere valido: un thread che entra nel monitor si aspetta di trovare la risorsa in uno *stato consistente*
- In tutti i tipi di monitor, *l'invariante di base* deve essere ripristinato:
  - prima di eseguire una cond.wait
  - prima di completare la procedura entry ed uscire dal monitor

Monitor in semantica *Hoare*

- Nel monitor *alla Hoare*, una condizione *più specifica* deve essere stata verificata anche prima di una cond.signal
- Ad es., per i produttori-consumatori deve valere: d-e > 0 prima di una cond.signal ad un thread consumatore
- Il thread *segnalato dipende criticamente* da questa condizione logica e non può eseguire se non è soddisfatta
- Soluzione di Hoare:
  - priorità al segnalato (vedremo come, nella realizzazione)
  - il thread verifica la propria condizione specifica tramite **if**
- E se invece il thread in attesa verificasse la propria condizione sotto un **while** ?

Monitor in semantica Mesa

- E se invece il thread in attesa verificasse la propria condizione sotto un **while** anziché **if** ?
- Con il test della condizione sotto **if** in stile Hoare si ha:
  - valutazione della condizione più semplice (1 sola volta vs. 2 o più con uso del while)
  - più cambi di contesto
  - problema dei risvegli spurii *(spurious wakeup)*
- → Ipotesi: nuova valutazione della condizione dopo il risveglio (quindi uso del **while**) per prevenire risvegli spurii, costa poco!
- Con uso del while, è naturale lasciare *proseguire il thread segnalante* e si riducono anche i context switch
- → un ventaglio di opportunità

Monitor in semantica Mesa

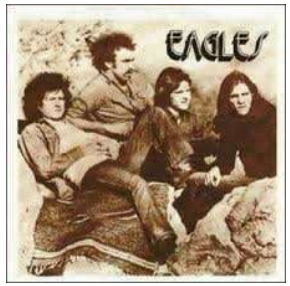
- Linguaggio Mesa (Lampson & Redell, 1980)
- Le primitive (principali) che operano sulle variabili condizione sono denominate *wait* e *notify*
- *Notify* significa che il thread bloccato sulla variabile condizione *potrebbe* poter proseguire! --> *hint*
- L'invariante specifico del thread segnalato non è garantito
- Il thread segnalato *deve comunque valutare nuovamente la condizione*, ed eventualmente ri-sospendersi (stile Mesa):

  while not ok_to_proceed *do* cond.wait *end;*
- anzichè (stile Hoare):

  if not ok_to_proceed *then* cond.wait;

Monitor in semantica Mesa

- Poichè il processo segnalante non subisce preemption:
  - meno context switch
  - migliore uso delle cache
  - non è necessario ristabilire l'*invariante* prima di effettuare *cond.notify* (come accade nel monitor alla Hoare)
  - protezione dal problema degli *spurious wakeup*
  - consente *time-out* sulle attese (*cond.timed_wait*), *broadcast*, "lassaise faire" programming (hint)
  - *broadcast* spesso denominata *cond.notifyAll*



- Il processo segnalato (“notificato”) verrà *prima o poi* posto in esecuzione, in base a decisioni di scheduling globali

Monitor in semantica Mesa

```
lock buf_lock = <initially unlocked>
condition producer_CV = <initialization function>
condition consumer_CV = <initialization function>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) { cond_wait(&producer_CV, &buf_lock); }
  enqueue(item);
  cond_signal(&consumer_CV);
  release(&buf_lock);
}
Consumer() {
  acquire(buf_lock);
  while (buffer empty) { cond_wait(&consumer_CV, &buf_lock); }
  item = dequeue();
  cond_signal(&producer_CV);
  release(buf_lock);
  return item
}
```

Monitor in semantica Mesa

- In generale, dopo la signal *il thread segnalato è stato semplicemente inserito nella coda dei thread pronti*
- Può trascorrere del tempo e la *condizione logica* per il suo progresso (il suo invariante specifico) può essere stata nel frattempo invalidata dal segnalante o da altri thread
- --> il thread deve verificare di nuovo la condizione logica di progresso (while) e riacquisire il lock
- Non è un vincolo sintattico, ma una buona pratica di programmazione; stili alternativi devono essere attentamente verificati sul caso concreto e concordati tra il team di sviluppatori
- Buone pratiche per *concurrent programming in the large*

Semantica Mesa e semantica Hoare

- Monitor in semantica Mesa significa *monitor in cui il segnalante mantiene il controllo del mutex* del monitor dopo avere segnalato la variabile condizione
- Resta ovviamente possibile ai thread (ma poco prudente) attendere sulle variabili condizione sotto if e non while, o anche in modo incondizionato; in tal caso, il segnalante deve garantire che sia verificato l'invariante specifico del thread in attesa, prima di effettuare la signal
- Anche nei monitor in semantica Hoare è possibile attendere sotto while; la semantica resta Hoare!

Tipi di monitor

- Semantica *signal and exit* → Concurrent Pascal
- Semantica *signal and wait* → Hoare
- Semantica *signal and continue* → Mesa, Java, Pthread

- Da notare che l'adozione di strumenti con semantica *signal and continue* si è consolidata in parallelo con la diffusione delle architetture multiprocessore e multicore